METHOD TO CREATE OPTIMIZED MACHINE CODE THROUGH COMBINED VERIFICATION AND TRANSLATION OF JAVA™ BYTECODE

By Inventor

Beat Heeb

CROSS REFERENCE TO RELATED APPLICATIONS

[0001.]This Application claims the benefit of U.S. Provisional Application No. 60/255,096 filed 12/13/2000, the disclosure of which is incorporated herein by reference.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH

[0002.]Not applicable.

BACKGROUND OF INVENTION

FIELD OF INVENTION

[0003.]The present invention is related to the combined compilation and verification of platform neutral bytecode computer instructions, such as JAVA. More specifically, the present invention relates to a new method of creating optimized machine code from platform neutral bytecode on either the development or target system by concurrently performing bytecode verification and compilation.

DESCRIPTION OF RELATED ART

[0004.]The benefit of architecture neutral language such as JAVA is the ability to execute such language on a wide range of systems once a suitable implementation technique, such as a JAVA Virtual Machine, is present. The key feature of the JAVA language is the creation and use of platform neutral bytecode instructions, which create the ability to run JAVA programs, such as applets, applications or servelets, on a broad range of diverse platforms. Typically, a JAVA program is compiled through the use of a JAVA Virtual Machine (JVM) which is merely an abstract computing machine used to compile the JAVA program (or source code) into platform neutral JAVA bytecode instructions, which are then placed into class files. The JAVA bytecode instructions in turn, serve as JVM instructions wherever the JVM is located. As bytecode instructions, the JAVA program may now be transferred to and executed by any system with a compatible JAVA platform. In addition, any other language which may be expressed in bytecode instructions, may be used with the JVM.

[0005.]Broadly speaking, computer instructions often are incompatible with other computer platforms. Attempts to improve compatibility include “high level” language software which is not executable without compilation into a machine specific code. As taught by U.S. Patent No. 5,590,331, issued December 31, 1996 to Lewis et al., several methods of compilation exist for this purpose. For instance, a pre-execution compilation approach may be used to convert “high level” language into machine specific code prior to execution. On the other hand, a runtime compilation approach may be used to convert instructions and immediately send the machine specific code to the processor for execution. A JAVA program requires a compilation step to create bytecode instructions, which are placed into class files. A class file contains streams of 8-bit bytes either alone or combined into larger values, which contain information about interfaces, fields or methods, the constant pool and the magic constant. Placed into class files, bytecode is an intermediate code, which is independent of the platform on which it is later executed. A single line of bytecode contains a one-byte opcode and either zero or additional bytes of operand information. Bytecode instructions may be used to control stacks, the VM register arrays or transfers. A JAVA interpreter is then used to execute the compiled bytecode instructions on the platform.

[0006.]The compilation step is accomplished with multiple passes through the bytecode instructions, where during each pass, a loop process is employed in which a method loops repeatedly through all the bytecode instructions. A single bytecode instruction is analyzed during each single loop through the program and after each loop, the next loop through the bytecode instructions analyzes the next single bytecode instruction. This is repeated until the last bytecode instruction is reached and the loop is ended.

[0007.]During the first compilation pass, a method loops repeatedly through all the bytecode instructions and a single bytecode instruction is analyzed during each single loop through the program. If it is determined the bytecode instruction being analyzed is the last bytecode instruction, the loop is ended. If the bytecode instruction being analyzed is not the last bytecode instruction, the method determines stack status from the bytecode instruction and stores this in stack status storage, which is updated for each bytecode instruction. This is repeated until the last bytecode instruction is reached and the loop is ended.

[0008.]During the second compilation pass, a method loops repeatedly through all the bytecode instructions once again and a single bytecode instruction is analyzed during each single loop through the program. If it is determined the bytecode instruction being analyzed is the last bytecode instruction, the loop is ended. If the bytecode instruction being analyzed is not the last bytecode instruction, the stack status storage and bytecode instruction are used to translate the bytecode instruction into machine code. This is repeated until the last bytecode instruction is translated and the loop is ended.

[0009.]A JAVA program however, also requires a verification step to ensure malicious or corrupting code is not present. As with most programming languages, security concerns are addressed through verification of the source code. JAVA applications ensure security through a bytecode verification process which ensures the JAVA code is valid, does not overflow or underflow stacks, and does not improperly use registers or illegally convert data types. The verification process traditionally consists of two parts achieved in four passes. First, verification performs internal checks during the first three passes, which are concerned solely with the bytecode instructions. The first pass checks to ensure the proper format is present, such as bytecode length. The second pass checks subclasses, superclasses and the constant pool for proper format. The third pass actually verifies the bytecode instructions. The fourth pass performs runtime checks, which confirm the compatibility of the bytecode instructions.

[0010.]As stated, verification is a security process, which is accomplished through several passes. The third pass in which actual verification occurs, employs a loop process similar to the compilation step in which a method loops repeatedly through all the bytecode instructions and a single bytecode instruction is analyzed during each single loop through the program. After each loop, the next loop through the bytecode instructions analyzes the next single bytecode instruction which is repeated until the last bytecode instruction is reached and the loop is ended.

[0011.]During the verification pass, the method loops repeatedly through all the bytecode instructions and a single bytecode instruction is analyzed during each single loop through the program. If it is determined the bytecode instruction being analyzed is the last bytecode instruction, the loop is ended. If the bytecode instruction is not the last bytecode instruction, the position of the bytecode instruction being analyzed is determined. If the bytecode instruction is at the beginning of a piece of code that is executed contiguously (a basic block), the global stack status is read from bytecode auxiliary data and stored. After storage, it is verified that the stored global stack status is compliant with the bytecode instruction. If however, the location of the bytecode instruction being analyzed is not at the beginning of a basic block, the global stack status is not read but is verified to ensure the global stack status is compliant with the bytecode instruction. After verifying that the global stack status is compliant with the bytecode instruction, the global stack status is changed according to the bytecode instruction. This procedure is repeated during each loop until the last bytecode instruction is analyzed and the loop ended.

[0012.]It may be noted that the pass through the bytecode instructions that is required for verification closely resembles the first compilation pass. Duplicate passes during execution can only contribute to the poor speed of JAVA programs, which in some cases may be up to 20 times slower than other programming languages such as C. The poor speed of JAVA programming is primarily the result of verification. In the past, attempts to improve speed have included compilation during idle times and pre-verification. In U.S. Patent No. 5,970,249 issued October 19,1999 to Holzle et al., a method is taught in which program compilation is completed during identified computer idle times. And in U.S. Patent No. 5,999,731 issued December 7, 1999 to Yellin et al. the program is pre-verified, allowing program execution without certain determinations such as stack overflow or underflow checks or data type checks. Both are attempts to improve execution speed by manipulation of the compilation and verification steps. In order to further improve speed, a method and apparatus is needed that can combine these separate, yet similar steps, the verification pass, and the first and second compilation pass, into a step which accomplishes the multiple tasks in substantially less time.

BRIEF SUMMARY OF THE INVENTION

[0013.]It is the object of the present invention to create a method and apparatus which may be used to combine compilation and verification of platform independent bytecode, either on the development system or within the target system, into optimized machine code thereby improving execution speed. Considering the required steps of bytecode compilation and verification, similarities between the two may be used to combine steps thereby reducing the time required to achieve both. The new method consists of a program instruction set which executes fewer passes through a bytecode instruction listing where complete verification and compilation is achieved, resulting in optimized machine code.

[0014.]The new method loops repeatedly through all the bytecode instructions and a single bytecode instruction is analyzed during each single loop through the program. If it is determined the bytecode instruction being analyzed is the last bytecode instruction, the loop is ended. If the bytecode instruction is not the last bytecode instruction however, the position of the bytecode instruction is determined and if the bytecode instruction being analyzed is at the beginning of a piece of code that is executed contiguously (a basic block), the global stack status is read from bytecode auxiliary data and stored. After storage, it is verified that the stored global stack status is compliant with the bytecode instruction. If however, the location of the bytecode instruction being analyzed is not at the beginning of a basic block, the global stack status is not read, but is verified to ensure the global stack status is compliant with the bytecode instruction. After verifying that the global stack status is compliant with the bytecode instruction, the global stack status is changed according to the bytecode instruction being analyzed. In addition, stack status is determined from the bytecode instruction being analyzed and stored in stack status storage. In doing so, the new method achieves complete verification and partial compilation (the steps traditionally performed during separate verification and compilation in the prior art).

[0015.]Next, the new method loops repeatedly through all the bytecode instructions and if it is determined the bytecode instruction being analyzed is the last bytecode instruction, the loop is ended, otherwise the pass is repeated for each bytecode listing within each class file. If the bytecode instruction is not the last bytecode instruction, the stack status storage and bytecode instruction are used to translate the bytecode instruction being analyzed into optimized machine code and this is repeated until the last bytecode instruction is translated and the loop is ended.

[0016.]The new method achieves complete verification and compilation of the bytecode instructions into optimized machine code on the development or target system. Through the combined steps, compilation and verification occur simultaneously using the new method.

BRIEF DESCRIPTION OF DRAWINGS

[0017.]These and other objects, features and characteristics of the present invention will become more apparent to those skilled in the art from a study of the following detailed description in conjunction with the appended claims and drawings, all of which form a part of this specification. In the drawings:

FIG. 1A (prior art) illustrates a flowchart of traditional bytecode instruction first pass compilation;

FIG. 1B (prior art) illustrates a flowchart of traditional bytecode instruction second pass compilation;

FIG. 2 (prior art) illustrates a flowchart of traditional bytecode instruction verification;

FIG. 3 illustrates a main flowchart of the embodiment of the new method;

FIG. 4A illustrates a subset flowchart of the embodiment of the new method; and

FIG. 4B further illustrates a subset flowchart of the embodiment of the new method.

DETAILED DESCRIPTION OF PRESENTLY PREFERRED EXEMPLARY EMBODIMENTS

[0018.]The present invention provides an improved method and apparatus to perform platform independent bytecode compilation and verification creating optimized machine code on an independent platform. The present invention, by creating a new bytecode compilation method combined with instruction verification, increases the speed and applicability of bytecode programming.

[00019.]In prior art Figures 1A and 1B, an illustrative flow diagram of traditional bytecode compilation is shown. In prior art Figure 1A, a traditional compilation method is shown as flow diagram 100 which loops through the bytecode instructions, analyzing an individual bytecode instruction during each loop as stated in step 102. After each bytecode instruction is analyzed, the method determines the stack status from the bytecode instruction being analyzed and stores the stack status in stack status storage as stated in step 104. When the last bytecode instruction is analyzed as stated step 102, the loop is ended at step108 and partial compilation is completed.

[0020.]In prior art Figure 1B, remaining compilation occurs in flow diagram 150 which shows further loops through the bytecode instructions analyzing an individual bytecode instruction during each loop as stated in step 152. The stack status storage and bytecode instruction are then used to translate the bytecode instruction into machine code as stated in step 154. When the last bytecode instruction is translated as stated in step 152, the loop is ended at step 158 and compilation is completed.

[0021.]In prior art Figure 2, an illustrative flow diagram of traditional bytecode verification is shown in flow diagram 200 which loops through the bytecode instructions, analyzing each until the last instruction is reached as stated in step 202. During each loop, the method analyzes a single bytecode instruction and if the method determines it has reached the last bytecode instruction, the loop is ended at step 214. Otherwise, the method determines the bytecode instruction position as stated in step 204. If the bytecode instruction being analyzed is at the beginning of a basic block, then the method reads the global stack status from bytecode auxiliary data and stores it as stated in step 206. After storage, the method verifies that the stored global stack status is compliant with the bytecode instruction as stated in step 208. If the bytecode instruction is not at the beginning of a basic block as stated in step 204, the global stack status is not read, but is verified to ensure the global stack status is compliant with the bytecode instruction as stated in step 208. In this case, step 206 is omitted. The global stack status is then changed according to the bytecode instruction as stated in step 210. This is repeated for each bytecode instruction until the last instruction is analyzed as stated in step 202 and the loop is ended at step 214.

[0022.]In Figures 3, 4A and 4B an illustrative flow diagram of the new method is shown. It may be noted from earlier prior art Figures that the pass through the bytecode instructions that is required for verification resembles compilation procedures. In the case of verification, the effect of the bytecode instruction on the stack must be analyzed and stored as a global stack status (i.e. a single storage location that is updated for every bytecode). This global storage stack must be filled from auxiliary data each time a basic block of data is entered. In the case of compilation, a similar analysis must be performed, however the stack status must be stored (in less detail) in stack status storage for each bytecode instruction analyzed.

[0023.]The present invention provides an improved method and apparatus to perform platform independent bytecode compilation and verification creating optimized machine code on an independent platform. The present invention creates a new method in which bytecode compilation is combined with instruction verification thereby increasing the speed and applicability of bytecode programming.

[0024.]Figure 3 is a main flowchart of a method 300 for combined bytecode verification and compilation in accordance with the new invention. In step 302, a class file placed on the development or target system is selected and a first method within the first class file is selected in step 304. At this point, the stack status for the first instruction and handler targets is set up in step 306. In step 308 a first bytecode instruction is selected and evaluated to determine if the instruction is setup in step 310. If the instruction is setup, the instruction is analyzed as outlined in Figures 4A and 4B. If the instruction is not setup, the next setup instruction is selected in step 312 and types are loaded from the stack map in step 314.

[0025.]Once the instruction has been analyzed in step 316, the following instruction is selected in step 318. If there are no remaining instructions as determined in step 320, the next method is selected in steps 322 and 328. If there are no remaining methods, the next class is selected in steps 324 and 330. If there are no remaining classes, the evaluation returns in step 326.

[0026.]Figures 4A and 4B are subset flowcharts of a method 400 for the analyses of each bytecode instruction from step 316 in Figure 3. In step 402, the selected instruction is checked to determine if it is within the scope of the exception handler. If it is, the compatibility between the actual local variable types and the exception handler stack map entry in bytecode is verified in step 404. If not, the instruction is set to "handled" in step 406 and the stack status of the actual instruction is copied to the new stack status.

[0027.]Next the instruction is evaluated to determine if there is a resulting pop from the stack in step 408 or a resulting push to the stack in step 414. If there is a resulting pop from the stack indicating an overflow condition, the compatibility between the stack status and expected values is verified in step 410 and the new stack status is then modified according to the instruction in step 412. If there is a resulting push to the stack indicating an underflow condition, the new stack status is modified according to the instruction and new actual stack types are set according to the instruction in step 416.

[0028.]In steps 418 and 422 the instruction is evaluated to determine if the instruction reads a local variable or writes to a local variable. If the instruction reads a local variable, the compatibility between the actual local variable type and the instruction is verified in step 420. If the instruction writes to a local variable, the variable type is modified according to the actual instruction.

[0029.]In step 426, the first successor instruction is evaluated. The instruction immediately following the actual instruction, determined in step 428, is dealt with in step 438 after all other successor instructions have been dealt with by step 436. Each successor instruction other than the instruction immediately following the actual instruction is evaluated in step 430 to determine if the instruction is marked as "none". If the successor instruction is marked as "none", the stack status of the successor instruction is initialized to the new stack status and the successor instruction is marked as "setup" in step 432 and the compatibility between the new stack status and the stack map for the successor instruction in the bytecode is verified. The compatibility between the actual stack, local variable types and stack map for the successor instruction is verified in step 434 and repeated until no further successor instructions remain.

[0030.]If the instruction is immediately following the actual instruction, step 438 determines if the instruction is a successor instruction and if so, step 440 determines if the instruction is marked as “none”. If the successor instruction is marked as “none”, the stack status of the following instruction is initialized to the new stack status and the following instruction is marked as “setup” in step 442. The compatibility between the new stack status and the stack map is verified. If there is a stack map for the successor instruction in step 444, the compatibility between the actual stack, local variable types and stack map for the successor instruction is verified in step 446 and types are loaded from the stack map in step 448. Once completed, step 450 returns to the main flowchart at step 318.

[0031.]Referring to Table 1, the new combined compilation and verification method places each class file in the development or target system, at which point each method in the class containing bytecode instructions is analyzed. The stack status for the first instruction and handler targets is setup. Temporary storage is created for stack status and marks for each bytecode instruction, in addition temporary storage for actual types of stack values and local variables is created.

[0032.]Next, the method initializes the stack status of the first instruction to empty and the stack status of the exception handler target instructions is initialized to contain the given exception. The marks of the first instruction and handler target instructions are set to “setup” and all other marks are set to “none”. The method signature is then used to initialize actual local variable types and the first bytecode instruction is set to be the actual instruction. This is repeated until no further instructions are marked as “setup”.

[0033.]The next subsequent bytecode instruction in turn which is marked as "setup" is set to be the actual instruction. The actual stack and local variable types from the stack map belonging to the actual instruction (each bytecode instruction) are loaded. If the actual instruction is within the scope of the exception handler, the compatibility between the actual local variable types and the exception handler stack map entry in bytecode is verified. Once verified or where the actual instruction is not within the scope of the exception handler, the selected bytecode instruction is set to "handled" and the stack status of the actual instruction is copied to new stack status.

[0034.]If the actual instruction pops one or more values from the stack, the compatibility between the stack status and expected values is verified and the new stack status is then modified according to the instruction. If the actual instruction pushes one or more values to the stack, the new stack status is modified according to the instruction and new actual stack types are set according to the instruction.

[0035.]A check for overflow and underflow conditions occurs next. If the actual instruction pops one or more values from the stack, check for underflow and verify the compatibility between the stack status and expected values and then modify the new stack status is according to the instruction. If there is no underflow condition, overflow conditions are evaluated. If the actual instruction pushes one or more values to the stack, check for overflow and modify the new stack status according to the instruction and new actual stack types are set according to the instruction.

[0036.]Once overflow and underflow checks are performed, the instruction is evaluated to determine if it reads a local variable or writes to a local variable. If the actual instruction reads a local variable, the compatibility between the actual local variable type and the instruction is verified. If the actual instruction writes to a local variable, the actual local variable type is modified according to the actual instruction.

[0037.]The first successor instruction is then evaluated. For each successor instruction except the one immediately following the actual instruction, if the successor instruction is marked as "none", the stack status of the successor instruction is initialized to the new stack status and the successor instruction is marked as "setup". The compatibility between the new stack status and the stack map for the successor instruction in the bytecode is verified. Once the successor is "setup", or if it was already "setup", the compatibility between the actual stack, local variable types and stack map for the successor instruction in the bytecode is also verified.

[0038.]If the instruction immediately following the actual instruction is a successor of the actual instruction and the following instruction is marked as "none", the stack status of the following instruction is initialized to the new stack status. The following instruction is then marked as “setup”. Once the successor is “setup”, or if it was already “setup”, if there is a stack map in the bytecode for the following instruction, the compatibility between new stack status and the stack map is verified. The compatibility between actual stack, local variable types and the stack map is also verified. The actual types are then loaded from the stack map and the actual instruction is changed to the immediately following instruction. The process is repeated for each method within each class file, and thereafter repeated for each class file.

[0039.]Prior art improvement methods in which computer idle time is filed with compilation steps and pre-verification, do not teach a method of combining verification and compilation steps. Also, idle time compilation is constantly subject to interruption and pre-verification may not eliminate all malicious code present. The result of using the new method shown in Figures 3, 4A, 4B and Table 1, is complete compilation and verification into optimized machine code with fewer program operations and reduced process times.